C11

1.智能指针

1.1智能指针解决的问题/优点

智能指针与普通指针用法相似，不同在于智能指针可以在适当时机自动释放分配的内存，**智能指针主要解决：**

(1)有些内存资源已经被释放，但指向它的指针并没有改变指向（成为了悬空指针）,但后续还在使用，由于悬空指针所指内存地址未知，可能会造成程序崩溃。
(2)有些内存资源已经被释放，后期又试图再释放一次（重复释放同一块内存会导致程序运行崩溃）；

(3)没有及时释放不再使用的内存资源，造成内存泄漏，程序占用的内存资源越来越多(new和delete没有配对)。

1.2 介绍智能指针

智能指针是一个模板类，它能存储指向动态分配对象的指针，负责自动释放动态分配的对象，防止堆内存泄漏。我们把动态分配的资源，交给一个类对象去管理，当类对象生命周期结束时，自动调用析构函数释放资源。智能指针就是这个原理

C++里面有四个智能指针，分别是auto_ptr、shared_ptr、unique_ptr、weak_ptr，auto_ptr在C++11中已被弃用，后三个是C++11支持的。

1.2.1 auto_ptr

auto_ptr构造时取得某个对象的控制权，在析构时释放该对象。实际是创建一个auto_ptr类型的局部对象，该局部对象析构时，会将自身所拥有的指针空间释放。(所以不会有内存泄漏)。

auto_ptr问题/被废弃原因：

(1)auto_ptr底层没有使用引用计数的方式，使用的是所有权模式，一个auto_ptr对象拥有它内部指针的所有权，当这个auto_ptr对象被释放的时候，对象的析构函数会自动调用delete来释放内部指针所指内存，所以当两个auto_ptr对象拥有同一个内部指针的所有权时，就可能会出现同一内存被重复释放的问题。

(2)另外auto_ptr的析构函数，删除对象用的是delete而不是delete[]，所以auto_ptr不能管理数组；

(3)当两个auto_ptr对象进行赋值操作时，内部指针被拥有的所有权会发生转移，这个赋值操作的右者，会丧失原内部指针的所有权，不再指向这个内部指针，而是指向空指针nullptr，如果程序运行时再去访问之前拥有这个内部指针的auto_ptr就会报错。

可以调用reset来重新分配指针的所有权，reset中会先释放原来的内部指针的内存，然后分配新的内部指针。

```
void foo_reset()
{
    //释放
    int* pNew = new int(3);
    int*p = new int(5);
    {
        std::auto_ptr<int> aptr(pNew);
        aptr.reset(p);
    }
}
```

1.2.2 unique_ptr原理 [juˈniːk]

unique_ptr采用的是独享所有权语义，一个非空的unique_ptr总是拥有它所指向的指针类型，不能进行拷贝操作，只能进行移动操作。

unique_ptr在功能上与auto_ptr相同，但unique_ptr比auto_ptr提升了安全性（它没有浅拷贝，不会重复释放），还支持对数组的使用。

(unique_ptr和shared_ptr的区别在于)，unique_ptr指针指向的堆内存不会与其它unique_ptr共享，每个 unique_ptr指针都独自拥有他所指堆内存空间的所有权，unique_ptr指针指向的堆内存空间的引用计数只能为1，一旦该unique_ptr指针放弃对所指堆内存空间的所有权，则该空间会被立即释放回收。如果两个unique_ptr同时指向一块内存（进行拷贝操作），编译时会报错。(因为unique_ptr的内部不允许拷贝构造和赋值运算符)

unique_ptr的创建有两种方式：

方式1：可以创建出空的 unique_ptr 指针

```
std::unique_ptr<int> p1();
std::unique_ptr<int> p2(nullptr);
```

方式2：创建unique_ptr指针同时，明确其指向

```
std::unique_ptr<int> p3(new int);
```

对unique_ptr的赋值操作：reset和move

可使用reset成员函数为unique_ptr分配新的指针所有权，使用reset，让unique_ptr指向另一个地址：其中 p 表示一个普通指针，如果p为nullptr，则当前 unique_ptr 也变成空指针；p不为nullptr，则该函数会释放当前 unique_ptr 指针指向的堆内存（如果有），然后获取 p 所指堆内存的所有权（p 为 nullptr）。

因为unique_ptr不能将自身对象内部指针直接赋值给其他unique_ptr，所以这里可以使用std::move()函数，让unique_ptr交出其内部指针的所有权，而自身置空，内部指针不会释放。

1.2.3 shared_ptr原理

是共享式智能指针，(可以实现多个智能指针共同使用同一块堆内存),它的底层采用的是引用计数方法，允许多个智能指针指向同一个对象，每多一个指针指向该对象，指向该对象的所有智能指针内部的引用计数加1，减少一个智能指针指向对象，引用计数会减1，当计数为0的时候会自动的释放动态分配的资源。

shared_ptr底层包含了两个成员，一个是内部的原始指针，指向shared_ptr对象，还有一个用来管理强弱引用计数的控制块(引用计数器类)，它也是一个指针，指向引用计数值，表示引用计数的那个变量是在堆上的，这样只要有一个智能指针增加了计数值，其余智能指针就可以通过指针访问到堆上同一计数值实现同步更新l了。

（shared_ptr 指针和它相关资源会在引用计数为0时被销毁释放。）

1.24 new和make_shared区别

使用make_shared生成的shared_ptr，优点在于底层只分配一次内存，对象资源和控制块分配在同一块内存中，而通过new指针创建出来的shared_ptr，要两次内存分配，对象资源分配一次，指针控制块分配一次；但是make_shared创建shared_ptr在强引用计数减为0，弱引用计数不为0时对象资源不会被释放，这是它的缺点，而new指针创建的shared_ptr在强引用计数减为0时一定释放对象资源，弱引用计数不会影响资源的释放时机。

1.2.5 weak_ptr原理

1. weak_ptr是一个弱智能指针，它本身是不具有普通内部指针的功能，它需要与 shared_ptr搭配使用，只是用来观察它对应的强指针 shared_ptr的使用次数，它不会影响所指堆内存空间的引用计数。(原理只答1即可)
2. 因此，这里弱指针的在使用上，实际上是一个特例，即不增加引用计数也能获取对象，因此，实际上在使用弱指针时，不能通过弱指针，直接访问内部指针的数据，而应该是先判断该弱指针所观察的强指针是否存在（调用expired()函数），如果存在，那么则使用lock()函数来获取一个新的shared_ptr来使用对应的内部指针。

   weak_ptr 类型指针并不会影响所指堆内存空间的引用计数。

1.5 C+与JAVA区别

JAVA有垃圾回收机制，编程人员不需要考虑内存管理问题，可以有效防止内存溢出问题。

C++在堆区开辟内存后，需要程序员自己手动释放，现在C++也在方面做了改进，增加了RAII机制。

1.6 何为RAII

Raii翻译过来是资源获取即初始化，它是C++中一种管理资源、避免内存泄漏的方法，C++中创建对象时会自动调用构造函数，对象超出作用域时会自动调用析构函数，RAII就是使用一个对象，在它构造的时候获取对应的资源，在对象生命周期内控制对资源的访问，在对象析构的时候释放构造时获取的资源。

1.7 shared_ptr的内存泄露解决

(1)shared_ptr发生循环引用计数时

循环引用计数是指，定义了两个类，两个类中都包含另外一个类的shared_ptr指针，作为自己的成员，同时两个类创建完对象以后，互相引用对方的对象指针来给自己成员中的智能指针赋值，这个时候就会产生循环引用计数问题，最后两个shared_ptr智能指针的引用计数没法清0，资源得不到释放。(它们各自的引用计数都是2，第一次是在各自创建对象调用构造函数初始化时有了一次引用计数，第二次是互相赋值时又增加了一次，当出了作用域以后，两个指针都会调用析构函数，引用计数减为1，A和B互相持有对方智能指针的一次引用，二者都不肯释放，都在等对方把引用计数结束掉，再结束自己，形成死锁)

解决办法:可以将其中一个shared_ptr改为weak_ptr，因为weak_ptr绑定到一个shared_ptr时不会改变shared_ptr的引用计数，这样就不会出现最后引用计数不为0，资源得不到释放的情况。

(2)多线程情况下使用shared_ptr可能会发生内存泄漏

当shared_ptr的指向发生改变时，它需要两步操作，第一步是改变它内部指针的指向，第二步是再将引用计数加1，引用计数的增减是原子操作(引用计数是一个临界资源)，但是这个整体操作不是，所以在多线程情况下这两步执行期间可能发生线程切换，导致两个shared_ptr

1.8 智能指针的选择

1.9 智能指针引用计数

C++ 智能指针底层是采用引用计数的方式实现的。简单的理解，智能指针在申请堆内存空间的同时，会为其配备一个整形值（初始值为 1），每当有新对象使用这个堆内存，该整形值 +1；相反，每当使用这个堆内存的对象被释放时，该整形值减 1。当堆空间对应的整形值为 0 时，就表明不再有对象使用它，该堆空间就会被释放掉。

1.10智能指针线程安全

智能指针不是线程安全的，因为虽然引用计数的增减操作是原子操作，但是它内部是有两步操作，先改变内部原始指针的指向，再改变引用计数器的值，在分别执行这两步操作时，可能中间会发生线程切换。

所以多线程环境下，同时读同一个shared_ptr对象是线程安全的，因为读不改变它的值，但是如果是多个线程对同一个shared_ptr对象进行写，则需要加锁。

2.右值引用

1.右值引用是指以引用传递的方式(非值传递)使用C++的右值。(底层有一个临时变量)

2.区分左值和右值

有名称的，可以取地址的表达式就是左值，相反就是右值

```
int a = 5;
5 = a; //错误，5 不能为左值
//其中，变量a就是一个左值，而字面量5就是一个右值。值得一提的是，C++ 中的左值也可以当做右值使用，例如：
int b = 10; //b是一个左值
a = b;//a、b都是左值，只不过将b可以当做右值使用
//以上面定义的变量a、b为例，a和b是变量名，且通过 &a 和 &b 可以获得他们的存储地址，因此a和b都是左值；反之，字面量5、10，它们既没有名称，也无法获取其存储地址,因此5、10都是右值。
```

右值引用和左值引用一样，定义时必须进行初始化，且只能使用右值进行初始化，右值引用使用"&&"来表示：

```
int num = 10;
//int && a = num;  //右值引用不能初始化为左值
int && a = 10;
```

右值引用还可以对右值进行修改。例如：

```
int && a = 10;//a是10的右值引用
a = 100;//对右值进行修改
cout << a << endl;
//程序输出结果为 100。
```

右值引用的应用场景主要就是移动语义和完美转发

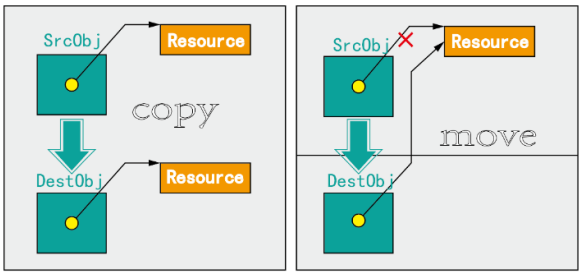
引用类型		可以引用的值类型			使用场景
	非常量左值	常量左值	非常量右值	常量右值	
非常量左值引用	Y	N	N	N	无
常量左值引用	Y	Y	Y	Y	常用于类中构建拷贝构造函数
非常量右值引用	N	N	Y	N	移动语义、完美转发
常量右值引用	N	N	Y	Y	无实际用途

3.移动语义

移动语义就是用来避免一些无畏的拷贝操作，移动语义的具体实现是移动构造，移动构造是以移动而不是深拷贝的方式初始化含有指针成员的类对象，对于含有指针成员的类对象，调用拷贝构造函数对其进行初始化时，为了防止出现堆区内存重复释放的问题，就需要使用深拷贝的方式，先在堆区申请一块和源对象同样大小的内存空间，再将源对象中的数据拷贝到目标对象中。使用移动构造的话，不会重新分配一块空间，而是直接接管源对象的数据，相当于是将自己的指针指向别人的资源，然后将别人的指针修改为nullptr。

二. 移动语义

(一) 深拷贝和移动的区别



copy和move的区别

1. 深拷贝：将SrcObj对象拷贝到DestObj对象，需要同时将Resourse资源也拷贝到DestObj对象去。这涉及到内存的拷贝。

2. 移动：通过“偷”内存的方式，将资源的所有权从一个对象转移到另一个对象上。但只是转移，并没有内存的拷贝。可见Resource的所有权只是从SrcObj对象转移到DestObj对象，由于不存在内存拷贝，其效率一般要高于复制构造。

4.完美转发

完美转发是指函数模板可以将自己的参数完美的转发给内部调用的其他函数(内层函数)，这里的完美就是说不仅可以准确的转发参数的值，还能保证被转发参数的左右值属性不变。

容器的emplace_back方法就是采用完美转发将参数转发给对象的构造函数，如果不用完美转发，参数都是按左值传递的，**内层函数要想对对象使用移动语义，则需按右值转发。**

完美转发中的两个点：万能引用和引用折叠

万能引用是指，C++11规定，一般情况下右值引用形式的参数只能接受右值，但对于函数模板中使用右值引用定义的参数，它既可以接收右值也可以接收左值。

引用折叠：

- 当实参为左值或者左值引用（A&）时，函数模板中 T&& 将转变为 A&（A& && = A&）；
- 当实参为右值或者右值引用（A&&）时，函数模板中 T&& 将转变为 A&&（A&& && = A&&）。

读者只需要知道，在实现完美转发时，只要函数模板的参数类型为 T&&，则 C++ 可以自行准确地判定出实际传入的实参是左值还是右值。

```
//实现完美转发的函数模板
template <typename T>
void function(T&& t) {
    otherdef(forward<T>(t));
}
```

总的来说，在定义模板函数时，我们采用右值引用的语法格式定义参数类型，由此该函数既可以接收外界传入的左值，也可以接收右值；其次，还需要使用 C++11 标准库提供的 forword() 模板函数修饰被调用函数中需要维持左、右值属性的参数。由此即可轻松实现函数模板中参数的完美转发。

5.初始化列表

初始化列表好处：(1)统一初始化方式，直接在变量名后面跟上初始化列表，来进行对象的初始化。

(2)如果构造函数的参数中有自定义的类型，使用初始化列表，会直接调用该成员变量对应的构造函数即完成初始化。如果是在构造函数中初始化，对象的成员变量会先调用默认构造函数为成员变量初始化，然后再调用该成员变量对应的构造函数再初始化一次。

必须使用初始化列表的时候:
1.常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
2.引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面

3.没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化

6.Lambda表达式

1. 利用lambda表达式可以编写内嵌的匿名函数，用以替换独立函数或者函数对象；
2. Lambda 表达式的定义形式如下：

```
[外部变量访问方式说明符] (参数表) mutable
noexcept/throw() -> 返回值类型
{
    语句块
}
//mutable
//此关键字可以省略，如果使用则之前的 () 小括号将不能省略（参数个数可以为 0）。默认情况下，对于以值传递方式引入的外部变量，不允许在 lambda 表达式内部修改它们的值（可以理解为这部分变量都是 const 常量）。而如果想修改它们，就必须使用
//noexcept/throw()
//可以省略，如果使用，在之前的 () 小括号将不能省略（参数个数可以为 0）。默认情况下，lambda 函数的函数体中可以抛出任何类型的异常。而标注 noexcept 关键字，则表示函数体内不会抛出任何异常；使用 throw() 可以指定 lambda 函数内部可以抛出的异常类型。

int a[4] = {11, 2, 33, 4};
sort(a, a+4, [=](int x, int y) -> bool { return x%10 < y%10; } );
for_each(a, a+4, [=](int x) { cout << x << " ";} );
```

4. lambda必须使用尾置返回来指定返回类型，可以忽略参数列表和返回值，但必须永远包含捕获列表和函数体；

智能指针是线程安全的吗?

·其实智能指针的引用计数是线程安全的，但是对于对象的访问并不是。也就是说**它控 制对象的消亡是线程安全的，但是对于对象的读写并不是。**

·同一个智能指针对象可以被多个线程读。

·不同的shared_ptr可以被多个线程修改，虽然它们管理同一个对象

·为了保证线程安全，如果我们对多个线程同时读写同一个shared_ptr对象，那么我们 需要加锁

如何判断weak_ptr的对象是否失效?

·调用expired()函数检查被引用的对象是否已删除。

·lock()会返回shared指针，判断该指针是否为空。

·use_count()也可以得到shared引用的个数，但速度较慢。

shared_ptr和unique_ptr区别?

·unique具有唯一性，对指向的对象值存在唯一的unique_ptr。unique_ptr不可复制， 赋值，但是move()可以转换对象的所有权，局部变量的返回值除外。与shared_ptr相比， 若自定义删除器，需要在声明处指定删除器类型，而shared不需要，shared自定义删除 器只需要指定删除器对象即可，在赋值时，可以随意赋值，删除器对象也会被赋值给新 的对象。

·原因：unique的实现中，删除器对象是作为unique_ptr的一部分，而shared_ptr，删 除器对象保存在control_block中。

7.NULL和nullptr

NULL:本质是宏定义，即 #define NULL 0

nullptr:是C++11中关键字，它是有类型的，使用nullptr可以提高代码的健壮性。